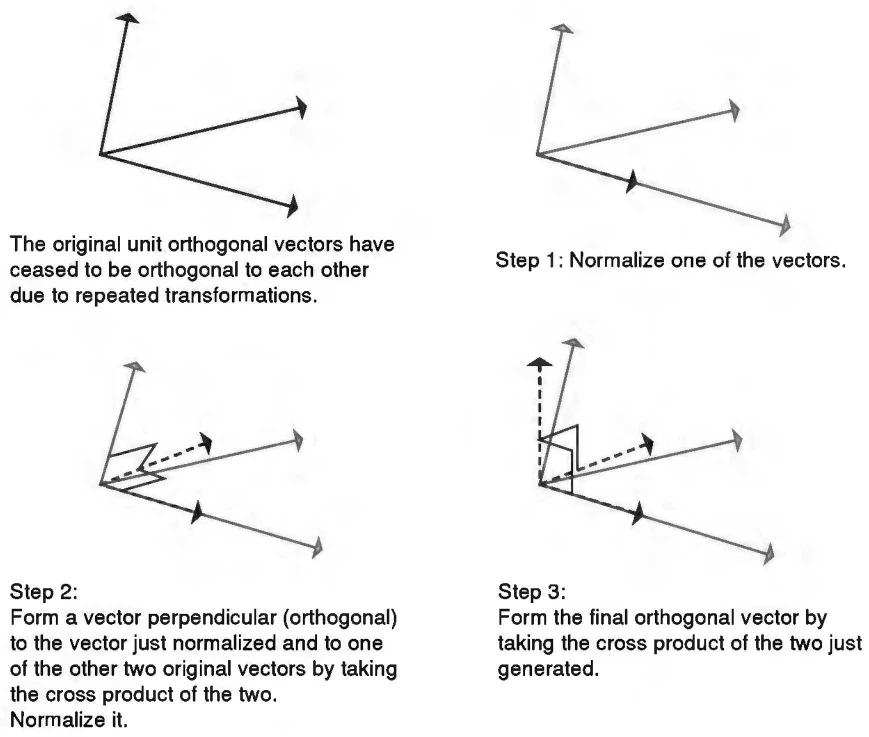


The original unit orthogonal vectors have ceased to be orthogonal to each other due to repeated transformations.

Step 1: Normalize one of the vectors.

Step 2:
Form a vector perpendicular (orthogonal) to the vector just normalized and to one of the other two original vectors by taking the cross product of the two.
Normalize it.

Step 3:
Form the final orthogonal vector by taking the cross product of the two just generated.

FIGURE 2.14

Orthonormalization of a set of three vectors.

2.2 Orientation representation

A common issue that arises in computer animation is deciding the best way to represent the position and orientation of an object in space and how to interpolate the represented transformations over time to produce motion. A typical scenario is one in which the user specifies an object in two transformed states and the computer is used to interpolate intermediate states, thus producing animated key-frame motion. Another scenario is when an object is to undergo two or more successive transformations and it would be efficient to concatenate these transformations into a single representation before applying it to a multitude of object vertices. This section discusses possible orientation representations and identifies strengths and weaknesses; the next chapter addresses the best way to interpolate orientations using these representations. In this discussion, it is assumed that the final transformation applied to the object is a result of rotations and translations only, so that there is no scaling involved, nonuniform or otherwise; that is, the transformations considered are *rigid body*.

The first obvious choice for representing the orientation and position of an object is by a 4×4 transformation matrix. For example, a user may specify a series of rotations and translations to apply to an object. This series of transformations is compiled into 4 × 4 matrices and is multiplied together to produce a compound 4 × 4 transformation matrix. In such a matrix, the upper left 3 × 3 submatrix represents a rotation to apply to the object, while the first three elements of the fourth column represent the translation (assuming points are represented by column vectors that are premultiplied by the transformation matrix). No matter how the 4 × 4 transformation matrix was formed (no matter in what order the transformations were given by the user, such as "rotate about *x*, translate, rotate about *x*, rotate about *y*, translate, rotate about *y*"), the final 4 × 4 transformation matrix produced by multiplying all of the individual transformation matrices in the specified order will result in a matrix that specifies the final position of the object by a 3 × 3 rotation matrix followed by a translation. The conclusion is that the rotation can be interpolated independently from the translation. (For now, consider that the interpolations are linear, although higher order interpolations are possible; see Appendix B.5.)

Consider two such transformations that the user has specified as key states with the intention of generating intermediate transformations by interpolation. While it should be obvious that interpolating the translations is straightforward, it is not at all clear how to go about interpolating the rotations. In fact, it is the objective of this discussion to show that interpolation of orientations is not nearly as straightforward as interpolation of translation. A property of 3 × 3 rotation matrices is that the rows and columns are orthonormal (unit length and perpendicular to each other). Simple linear interpolation between the nine pairs of numbers that make up the two 3 × 3 rotation matrices to be interpolated will not produce intermediate 3 × 3 matrices that are orthonormal and are therefore not rigid body rotations. It should be easy to see that interpolating from a rotation of +90 degrees about the *y*-axis to a rotation of −90 degrees about the *y*-axis results in intermediate transformations that are nonsense (Figure 2.15).

So, direct interpolation of transformation matrices is not acceptable. There are alternative representations that are more useful than transformation matrices in performing such interpolations including fixed angle, Euler angle, axis–angle, quaternions, and exponential maps.

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Positive 90-degree *y*-axis rotation Negative 90-degree *y*-axis rotation

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Interpolated matrix halfway between the orientation representations above

FIGURE 2.15

Direct interpolation of transformation matrix values can result in nonsense—transformations.

2.2.1 Fixed-angle representation

A *fixed-angle* representation[4] really refers to "angles used to rotate about fixed axes." A fixed order of three rotations is implied, such as *x*-*y*-*z*. This means that orientation is given by a set of three ordered parameters that represent three ordered rotations about fixed axes: first around *x*, then around *y*, and then around *z*. There are many possible orderings of the rotations, and, in fact, it is not necessary to use all three coordinate axes. For example, *x*-*y*-*x* is a feasible set of rotations. The only orderings that do not make sense are those in which an axis immediately follows itself, such as in *x*-*x*-*y*. In any case, the main point is that the orientation of an object is given by three angles, such as (10, 45, 90). In this example, the orientation represented is obtained by rotating the object first about the *x*-axis by 10 degrees, then about the *y*-axis by 45 degrees, and then about the *z*-axis by 90 degrees. In Figure 2.16, the aircraft is shown in its initial orientation and in the orientation represented by the values of (10, 45, 90).

The following notation will be used to represent such a sequence of rotations: $R_z(90)R_y(45)R_x(10)$ (in this text, transformations are implemented by premultiplying column vectors by transformation matrices; thus, the rotation matrices appear in right to left order).

From this orientation, changing the *x*-axis rotation value, which is applied first to the data points, will make the aircraft's nose dip more or less in the *y*-*z* plane. Changing the *y*-axis rotation will change the amount the aircraft, which has been rotated around the *x*-axis, rotates out of the *y*-*z* plane. Changing the *z*-axis rotation value, the rotation applied last, will change how much the twice-rotated aircraft will rotate about the *z*-axis.

The problem with using this scheme is that two of the axes of rotation can effectively line up on top of each other when an object can rotate freely in space (or around a 3 degree of freedom[5] joint). Consider an object in an orientation represented by (0, 90, 0), as shown in Figure 2.17. Examine the effect a slight change in the first and third parametric values has on the object in that orientation. A slight change of the third parameter will rotate the object slightly about the global *z*-axis because that is the rotation applied last to the data points. However, note that the effect of a slight change of the first

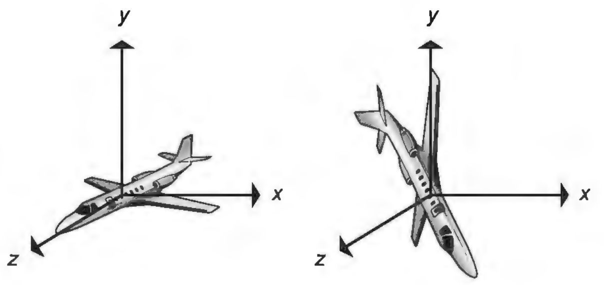


FIGURE 2.16

Fixed-angle representation.

[4]Terms referring to rotational representations are not used consistently in the literature. This book follows the usage found in *Robotics* [1], where fixed angle refers to rotation about the fixed (global) axes and Euler angle refers to rotation about the rotating (local) axes.

[5]The degrees of freedom that an object possesses is the number of independent variables that have to be specified to completely locate that object (and all of its parts).

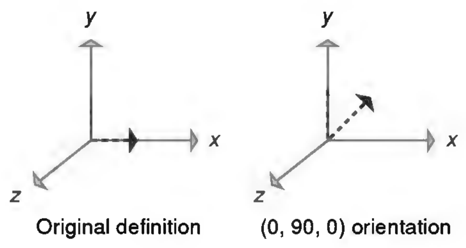


FIGURE 2.17

Fixed-angle representation of (0, 90, 0).

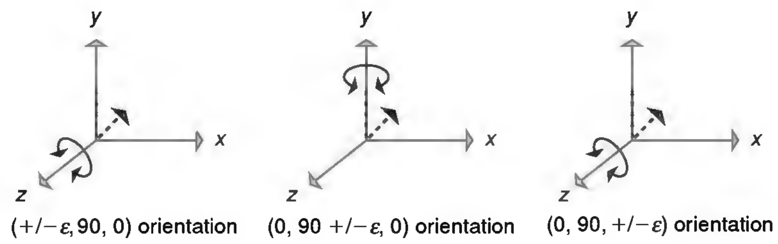

y
x
z
(+/−ε, 90, 0) orientation
y
x
z
(0, 90 +/−ε, 0) orientation
y
x
z
(0, 90, +/−ε) orientation

FIGURE 2.18

Effect of slightly altering values of fixed-angle representation (0, 90, 0).

parameter, which rotates the original data points around the x-axis, will also have the effect of rotating the transformed object slightly about the z-axis (Figure 2.18). This results because the 90-degree y-axis rotation has essentially made the first axis of rotation align with the third axis of rotation. The effect is called *gimbal lock*. From the orientation (0, 90, 0), the object can no longer be rotated about the global x-axis by a small change in its orientation representation. Actually, the representation that will perform an incremental rotation about the x-axis from the (0, 90, 0) orientation is $(90, 90 + \varepsilon, 90)$, which is not very intuitive.

The cause of this problem can often make interpolation between key positions problematic. Consider the key orientations (0, 90, 0) and (90, 45, 90), as shown in Figure 2.19. The second orientation is a

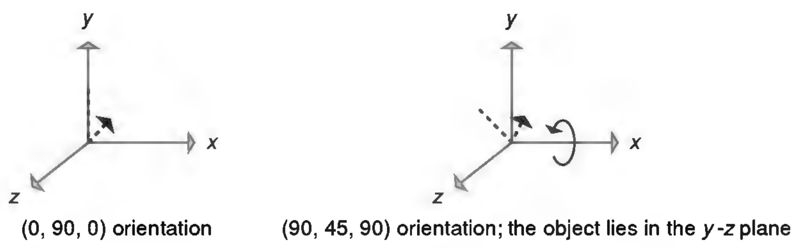


FIGURE 2.19

Example orientations to interpolate.

45-degree x-axis rotation from the first position. However, as discussed above, the object can no longer directly rotate about the x-axis from the first key orientation because of the 90-degree y-axis rotation. Direct interpolation of the key orientation representations would produce (45, 67.5, 45) as the halfway orientation, which is very different from the (90, 22.5, 90) orientation that is desired (because that is the representation of the orientation that is intuitively halfway between the two given orientations). The result is that the object will swing out of the y-z plane during the interpolation, which is not the behavior one would expect.

In its favor, the fixed-angle representation is compact, fairly intuitive, and easy to work with because the implied operations correspond to what we know how to do mathematically—rotate data around the global axes. However, it is often not the most desirable representation to use because of the gimbal lock problem.

2.2.2 Euler angle representation

In a *Euler angle* representation, the axes of rotation are the axes of the local coordinate system that rotate with the object, as opposed to the fixed global axes. A typical example of using Euler angles is found in the roll, pitch, and yaw of an aircraft (Figure 2.20).

As with the fixed-angle representation, the Euler angle representation can use any of various orderings of three axes of rotation as its representation scheme. Consider a Euler angle representation that uses an x-y-z ordering and is specified as (α, β, γ). The x-axis rotation, represented by the transformation matrix $R_x(\alpha)$, is followed by the y-axis rotation, represented by the transformation matrix $R_y(\beta)$, around the y-axis of the local, rotated coordinate system. Using a prime symbol to represent rotation about a rotated frame and remembering that points are represented as column vectors and are premultiplied by transformation matrices, one achieves a result of $R_y'(\beta)R_x(\alpha)$. Using global axis rotation matrices to implement the transformations, the y-axis rotation around the rotated frame can be effected by $R_x(\alpha)R_y(\beta)R_x(-\alpha)$. Thus, the result after the first two rotations is shown in Equation 2.22.

$$R_y'(\beta)R_x(\alpha) = R_x(\alpha)R_y(\beta)R_x(-\alpha)R_x(\alpha) = R_x(\alpha)R_y(\beta) \tag{2.22}$$

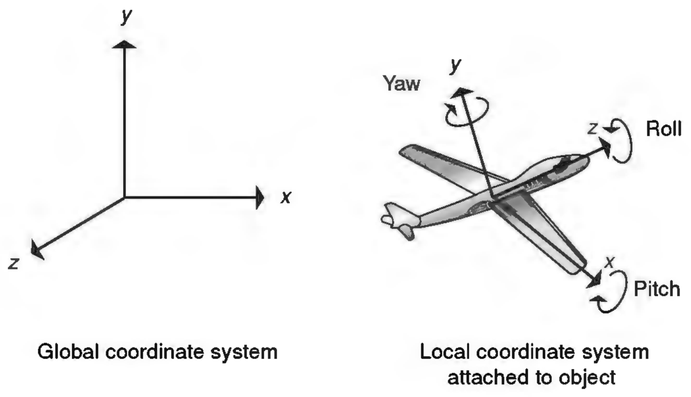


FIGURE 2.20

Euler angle representation.

The third rotation, $R_z(\gamma)$, is around the now twice-rotated frame. This rotation can be effected by undoing the previous rotations with $R_x(-\alpha)$ followed by $R_y(-\beta)$, then rotating around the global z-axis by $R_z(\gamma)$, and then reapplying the previous rotations. Putting all three rotations together, and using a double prime to denote rotation about a twice-rotated frame, results in Equation 2.23.

$$\begin{aligned} R_y''(\gamma)R_y'(\beta)R_x(\alpha) &= R_x(\alpha)R_y(\beta)R_z(\gamma)R_y(-\beta)R_x(-\alpha)R_x(\alpha)R_y(\beta) \\ &= R_x(\alpha)R_y(\beta)R_z(\gamma) \end{aligned} \tag{2.23}$$

Thus, this system of Euler angles is precisely equivalent to the fixed-angle system in reverse order. This is true for any system of Euler angles. For example, z-y-x Euler angles are equivalent to x-y-z fixed angles. Therefore, the Euler angle representation has exactly the same advantages and disadvantages (i.e., gimbal lock) as those of the fixed-angle representation.

2.2.3 Angle and axis representation

In the mid-1700s, Leonhard Euler showed that one orientation can be derived from another by a single rotation about an axis. This is known as the Euler Rotation Theorem [1]. Thus, any orientation can be represented by three numbers: two for the axis, such as longitude and latitude, and one for the angle (Figure 2.21). The axis can also be represented (somewhat inefficiently) by a three-dimensional vector. This can be a useful representation. Interpolation between representations (A_1, θ_1) and (A_2, θ_2), where A is the axis of rotation and θ is the angle, can be implemented by interpolating the axes of rotation and the angles separately (Figure 2.22). An intermediate axis can be determined by rotating one axis partway toward the other. The axis for this rotation is formed by taking the cross product of two axes, A_1 and A_2. The angle between the two axes is determined by taking the inverse cosine of the dot product of normalized versions of the axes. An interpolant, k, can then be used to form an intermediate axis and angle pair. Note that the axis–angle representation does not lend itself to easily concatenating a series of rotations. However, the information contained in this representation can be put in a form in which these operations are easily implemented: quaternions.

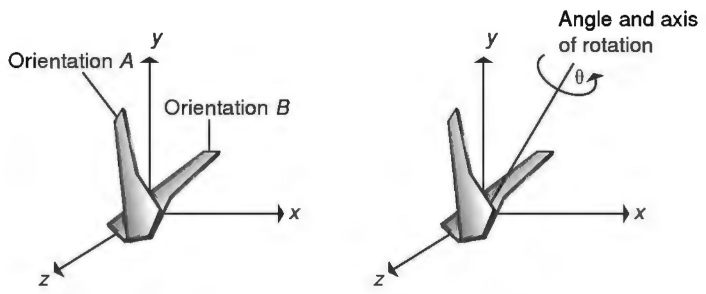
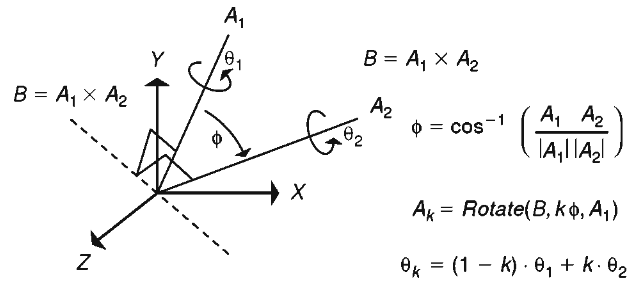


FIGURE 2.21

Euler's rotation theorem implies that for any two orientations of an object, one can be produced from the other by a single rotation about an arbitrary axis.



A_1
Y
θ_1
$B = A_1 \times A_2$
ϕ
A_2
θ_2
X
Z
$B = A_1 \times A_2$
$\phi = \cos^{-1}\left(\frac{A_1 \; A_2}{|A_1||A_2|}\right)$
$A_k = Rotate(B, k\phi, A_1)$
$\theta_k = (1 - k)\cdot\theta_1 + k\cdot\theta_2$

FIGURE 2.22

Interpolating axis-angle representations of (A_1, θ_1) and (A_2, θ_2) by k to get (A_k, θ_k), where 'Rotate(a,b,c)' rotates 'c' around 'a' by 'b' degrees.

2.2.4 Quaternion representation

As discussed earlier, the representations covered so far have drawbacks when interpolating intermediate orientations when an object or joint has three degrees of rotational freedom. A better approach is to use *quaternions* to represent orientation [5]. A quaternion is a four-tuple of real numbers, $[s, x, y, z]$, or, equivalently, $[s, v]$, consisting of a scalar, s, and a three-dimensional vector, v.

The quaternion is an alternative to the axis and angle representation in that it contains the same information in a different, but mathematically convenient, form. Importantly, it is in a form that can be interpolated as well as used in concatenating a series of rotations into a single representation. The axis and angle information of a quaternion can be viewed as an orientation of an object relative to its initial object space definition, or it can be considered as the representation of a rotation to apply to an object definition. In the former view, being able to interpolate between represented orientations is important in generating key-frame animation. In the latter view, concatenating a series of rotations into a simple representation is a common and useful operation to perform to apply a single, compound transformation to an object definition.

Basic quaternion math

Before interpolation can be explained, some basic quaternion math must be understood. In the equations that follow, a bullet operator represents dot product, and "×" denotes cross-product. *Quaternion addition* is simply the four-tuple addition of quaternion representations, $[s_1, v_1] + [s_2, v_2] = [s_1 + s_2, v_1 + v_2]$. *Quaternion multiplication* is defined as Equation 2.24. Notice that quaternion multiplication is associative, $(q_1\, q_2)q_3 = q_1(q_2\, q_3)$, but is not commutative, $q_1\, q_2 \neq q_2\, q_1$.

$$[s_1, v_1]\,[s_2, v_2] = [s_1 s_2 - v_1 \cdot v_2, s_1 v_2 + s_2 v_1 + v_1 \times v_2] \tag{2.24}$$

A point in space, v, or, equivalently, the vector from the origin to the point, is represented as $[0, v]$. It is easy to see that quaternion multiplication of two orthogonal vectors ($v_1 \bullet v_2 = 0$) computes the cross-product of those vectors (Eq. 2.25).

$$[0, v_1]\,[0, v_2] = [0, v_1 \times v_2] \text{ if } v_1 \cdot v_2 = 0 \tag{2.25}$$

The quaternion $[1, (0, 0, 0)]$ is the multiplicative identity; that is,

$$[s, \; v]\,[1, (0, 0, 0)] = [s, v] \tag{2.26}$$

The *inverse of a quaternion*, $[s, v]^{-1}$, is obtained by negating its vector part and dividing both parts by the magnitude squared (the sum of the squares of the four components), as shown in Equation 2.27.

$$q^{-1} = (1/||q||)^2[s, -v] \qquad \text{where } ||q|| = \sqrt{s^2 + x^2 + y^2 + z^2} \tag{2.27}$$

Multiplication of a quaternion, q, by its inverse, q^{-1}, results in the multiplicative identity [1, (0, 0, 0)]. A unit-length quaternion (also referred to here as a unit quaternion), $\hat{q}$, is created by dividing each of the four components by the square root of the sum of the squares of those components (Eq. 2.28).

$$\hat{q} = q/(||q||) \tag{2.28}$$

Representing rotations using quaternions

A rotation is represented in a quaternion form by encoding axis–angle information. Equation 2.29 shows a unit quaternion representation of a rotation of an angle, *u*, about a unit axis of rotation (*x*, *y*, *z*).

$$QuatRot(\theta, (x, y, z)) \equiv [\cos(\theta/2), \sin(\theta/2)(x, y, z)] \tag{2.29}$$

Notice that rotating some angle around an axis is the same as rotating the negative angle around the negated axis. In quaternions, this is manifested by the fact that a quaternion, $q = [s, v]$, and its negation, $-q = [-s, -v]$, represent the same rotation. The two negatives in this case cancel each other out and produce the same rotation. In Equation 2.30, the quaternion *q* represents a rotation of *u* about a unit axis of rotation (*x*, *y*, *z*), i.e.,

$$\begin{aligned} -q &= [-\cos(\theta/2), -\sin(\theta/2)(x, y, z)] \\ &= [\cos(180-(\theta/2)), \sin(\theta/2)(-(x, y, z))] \\ &= [\cos((360-\theta)/2), \sin(180-\theta/2)(-(x, y, z))] \\ &= [\cos((360-\theta)/2), \sin(360-\theta/2)(-(x, y, z))] \\ &\equiv QuatRot(-\theta, -(x, y, z)) \\ &\equiv QuatRot(\theta, (x, y, z)) \end{aligned} \tag{2.30}$$

Negating *q* results in a negative rotation around the negative of the axis of rotation, which is the same rotation represented by *q* (Eq. 2.30).

Rotating vectors using quaternions

To rotate a vector, *v*, using quaternion math, represent the vector as [0, *v*] and represent the rotation by a quaternion, *q*. The vector is rotated according to Equation 2.31.

$$Rot_q(v) \equiv qvq^{-1} = v' \tag{2.31}$$

A series of rotations can be concatenated into a single representation by quaternion multiplication. Consider a rotation represented by a quaternion, *p*, followed by a rotation represented by a quaternion, *q*, on a vector, *v* (Eq. 2.32).

$$Rot_q(Rot_p(v)) = q(pvp^{-1})q^{-1} = (qp)v(qp)^{-1} = Rot_{qp}(v) \tag{2.32}$$

The inverse of a quaternion represents rotation about the same axis by the same amount but in the reverse direction. Equation 2.33 shows that rotating a vector by a quaternion, *q*, followed by rotating the result by the inverse of that same quaternion produces the original vector.

$$Rot_{q^{-1}}(Rot_q(v)) = q^{-1}(qvp^{-1})q = v \tag{2.33}$$

Also, notice that in performing rotation, qvq^{-1}, all effects of magnitude are divided out due to the multiplication by the inverse of the quaternion. Thus, any scalar multiple of a quaternion represents the same rotation as the corresponding unit quaternion (similar to how the homogeneous representation of points is scale invariant).

A concise list of quaternion arithmetic and conversions to and from other representations can be found in Appendix B.3.4.

2.2.5 Exponential map representation

Exponential maps, similar to quaternions, represent an orientation as an axis of rotation and an associated angle of rotation as a single vector [3]. The direction of the vector is the axis of rotation and the magnitude is the amount of rotation. In addition, a zero rotation is assigned to the zero vector, making the representation continuous at the origin. Notice that an exponential map uses three parameters instead of the quaternion's four. The main advantage is that it has well-formed derivatives. These are important, for example, when dealing with angular velocity.

This representation does have some drawbacks. Similar to Euler angles, it has singularities. However, in practice, these can be avoided. Also, it is difficult to concatenate rotations using exponential maps and is best done by converting to rotation matrices.

2.3 Summary

Linear transformations represented by 4×4 matrices are a fundamental operation in computer graphics and animation. Understanding their use, how to manipulate them, and how to control round-off error is an important first step in mastering graphics and animation techniques.

There are several orientation representations to choose from. The most robust representation of orientation is quaternions, but fixed angle, Euler angle, and axis–angle are more intuitive and easier to implement. Fixed angles and Euler angles suffer from gimbal lock and axis–angle is not easy to composite, but they are useful in some situations. Exponential maps also do not concatenate well but offer some advantages when working with derivatives of orientation. Appendix B.3.4 contains useful conversions between quaternions and other representations.

References

[1] Craig J. Robotics. New York: Addison-Wesley; 1989.
[2] Foley J, van Dam A, Feiner S, Hughes J. Computer Graphics: Principles and Practice. 2nd ed. New York: Addison-Wesley; 1990.
[3] Grassia FS. Practical Parameterization of Rotations Using the Exponential Map. The Journal of Graphics Tools 1998;3.3.
[4] Mortenson M. Geometric Modeling. New York: John Wiley & Sons; 1997.
[5] Shoemake K. Animating Rotation with Quaternion Curves. In: Barsky BA, editor. Computer Graphics. Proceedings of SIGGRAPH 85, vol. 19(3). San Francisco, Calif; August 1985. p. 143–52.
[6] Strong G. Linear Algebra and Its Applications. New York: Academic Press; 1980.
[7] Watt A, Watt M. Advanced Animation and Rendering Techniques. New York: Addison-Wesley; 1992.